**1. what is python?**

Python is a high-level, interpreted programming language that is known for its readability, simplicity, and versatility. It was created in the late 1980s by Guido van Rossum and has since become one of the most popular programming languages in the world.

Python is often used for a wide variety of applications, including web development, data analysis, machine learning, scientific computing, and many other fields. Its simplicity and ease of use make it a popular choice for beginners, while its extensive libraries and frameworks make it a powerful tool for experienced developers.

Python is an open-source language, which means that its source code is freely available and can be modified and distributed by anyone. It has a large and active community of developers who contribute to the language and its ecosystem of libraries, tools, and frameworks.

Python code is typically written in text files with a .py extension, and can be run using the Python interpreter. The interpreter executes the code line by line and displays the output to the user. Python also has a rich set of built-in functions and modules, which makes it easy to perform common tasks like string manipulation, file I/O, and networking.

Overall, Python is a powerful, versatile, and easy-to-learn programming language that is widely used in many industries and fields.

**2. who develop python?**

Python was created in the late 1980s by Guido van Rossum, a Dutch computer programmer. He started working on the language in December 1989, while working at the National Research Institute for Mathematics and Computer Science (CWI) in the Netherlands.

Van Rossum remained the primary developer of Python for many years, and he continued to oversee its development and direction until stepping down as its Benevolent Dictator for Life (BDFL) in July 2018. Today, Python's development is overseen by the Python Software Foundation, a non-profit organization that supports and promotes the language and its community.

Python's development is largely driven by a community of contributors, who contribute code, documentation, and other resources to the language and its ecosystem of libraries, tools, and frameworks. This community is open and inclusive, and anyone can contribute to the language or its ecosystem, regardless of their experience or background.

**3. what is interpreter?**

An interpreter is a program that reads and executes code directly, without the need for a separate compilation step. Interpreted languages like Python are designed to be executed by an interpreter.

When you write Python code, you save it in a text file with a .py extension. The Python interpreter reads this file line by line and executes the code as it goes. This means that you can write and test code quickly, without the need to compile it first.

The Python interpreter is responsible for translating your code into machine-readable instructions that can be executed by the computer's processor. It also handles tasks like memory management and error handling.

Because Python is an interpreted language, it can be slower than compiled languages like C or C++. However, the ease of use and flexibility of interpreted languages make them popular choices for many applications, especially for tasks like data analysis, scientific computing, and prototyping.

Overall, an interpreter is a crucial component of interpreted programming languages like Python, and it allows developers to write and execute code quickly and easily.

**4. What is variable in Python?**

In Python, a variable is a name that is used to refer to a value in memory. Variables are used to store data, and their value can be changed as the program runs.

In Python, you can assign a value to a variable using the assignment operator (=). For example, the following code assigns the value 10 to a variable named x:

x = 10

Once a variable has been assigned a value, you can use it in expressions and operations. For example, you can add two variables together like this:

x = 10

y = 5

z = x + y

print(z)

This code assigns the value 10 to the variable x, the value 5 to the variable y, and then adds them together and assigns the result to the variable z. The print() function is used to display the value of z to the console.

In Python, variables can store many different types of data, including integers, floating-point numbers, strings, lists, tuples, and dictionaries. You don't need to declare the type of a variable before you use it - Python automatically assigns the appropriate type based on the value you assign to it.

Overall, variables are a fundamental concept in Python programming, and they allow you to store and manipulate data in your programs.

**5. what is operator in python?**

In Python, an operator is a symbol or keyword that is used to perform an operation on one or more values. Operators can be used to perform arithmetic, logical, and comparison operations, among other things.

**6. What are the types of operator in python?**

In Python, there are several types of operators that you can use to manipulate values in your programs. Here's an overview of the different types of operators:

**Arithmetic Operators**: Arithmetic operators are used to perform mathematical operations on numerical values. These operators include + (addition), - (subtraction), \* (multiplication), / (division), % (modulus), and \*\* (exponentiation).

**Comparison Operators:** Comparison operators are used to compare two values and return a Boolean value (True or False) based on the result. These operators include == (equal to), != (not equal to), < (less than), > (greater than), <= (less than or equal to), and >= (greater than or equal to).

**Logical Operators:** Logical operators are used to combine Boolean expressions and return a Boolean result. These operators include and (logical AND), or (logical OR), and not (logical NOT).

**Assignment Operators:** Assignment operators are used to assign a value to a variable. These operators include = (assign value to variable), += (add value to variable and assign result), -= (subtract value from variable and assign result), \*= (multiply variable by value and assign result), /= (divide variable by value and assign result), %= (take modulus of variable with value and assign result), and \*\*= (raise variable to power of value and assign result).

**Bitwise Operators:** Bitwise operators are used to perform bit-level operations on binary values. These operators include & (bitwise AND), | (bitwise OR), ^ (bitwise XOR), ~ (bitwise NOT), << (bitwise left shift), and >> (bitwise right shift).

**Membership Operators:** Membership operators are used to test if a value is a member of a sequence (such as a list or tuple). These operators include in (tests for membership) and not in (tests for non-membership).

**Identity Operators:** Identity operators are used to test if two values are the same object. These operators include is (tests for object identity) and is not (tests for object non-identity).

Overall, operators are a fundamental part of Python programming, and they allow you to perform a wide range of operations on values in your programs.

**7. what is membership operator in python?**

In Python, membership operators are used to test whether a value is a member of a sequence (such as a list, tuple, or string). The two membership operators are in and not in.

The in operator returns True if the specified value is found in the sequence, and False otherwise. Here's an example that uses the in operator to check if a value is in a list:

**Program:-**

**fruits = ['apple', 'banana', 'orange']**

**if 'banana' in fruits:**

**print("I found a banana!")**

**output:-**

I found a banana!

The not in operator, as the name suggests, returns True if the specified value is not found in the sequence, and False otherwise. Here's an example that uses the not in operator to check if a value is not in a tuple:

**Program:-**

**numbers = (1, 2, 3, 4, 5)**

**if 6 not in numbers:**

**print("6 is not in the tuple.")**

**Output:-**

**6 is not in the tuple.**

Membership operators are useful when you need to check if a value is present or absent in a sequence. They are commonly used in conditional statements to perform different actions based on whether a value is in a sequence or not.

**8. what is identity operator?**

In Python, identity operators are used to compare the memory location of two objects to determine if they are the same object. The two identity operators are is and is not.

The is operator returns True if both operands refer to the same object in memory, and False otherwise. Here's an example that uses the is operator to compare two string objects:

**Program:-**

**x = "hello"**

**y = "hello"**

**z = "world"**

**print(x is y) # True**

**print(x is z) # False**

The is not operator, as the name suggests, returns True if the operands do not refer to the same object in memory, and False otherwise. Here's an example that uses the is not operator to compare two list objects:

**Program:-**

**a = [1, 2, 3]**

**b = [1, 2, 3]**

**c = a**

**print(a is b) # False**

**print(a is not b) # True**

**print(a is c) # True**

In this example, the a and b lists have the same values, but they are different objects in memory, so the is operator returns False. However, the a and c lists refer to the same object in memory, so the is operator returns True.

Identity operators are useful when you need to compare objects based on their memory location rather than their values. They are commonly used to optimize code by reusing objects instead of creating new ones.

**9. what are arithmetic operator having?**

In Python, arithmetic operators are used to perform mathematical operations on numeric values. The arithmetic operators are:

+ (addition): Adds two operands

- (subtraction): Subtracts the second operand from the first operand

\* (multiplication): Multiplies two operands

/ (division): Divides the first operand by the second operand

% (modulus): Returns the remainder of the division of the first operand by the second operand

\*\* (exponentiation): Raises the first operand to the power of the second operand

// (floor division): Returns the floor of the division of the first operand by the second operand

Here's an example that uses all the arithmetic operators:

**Program:-**

**x = 10**

**y = 3**

**# addition**

**print(x + y) # 13**

**# subtraction**

**print(x - y) # 7**

**# multiplication**

**print(x \* y) # 30**

**# division**

**print(x / y) # 3.3333333333333335**

**# modulus**

**print(x % y) # 1**

**# exponentiation**

**print(x \*\* y) # 1000**

**# floor division**

**print(x // y) # 3**

Arithmetic operators are used in mathematical calculations, and they are essential in many programming tasks that involve numeric values.

**10. What is Feature of List?**

In Python, a list is a collection of ordered and mutable elements that can be of any data type. Lists are defined by enclosing a comma-separated sequence of elements within square brackets [ ]. Here are some of the key features of lists:

**1)Ordered**: Lists maintain the order of elements as they are added, and you can access individual elements by their index (position) in the list. For example:

**Program:-**

**fruits = ["apple", "banana", "orange"]**

**print(fruits[0]) # "apple"**

**print(fruits[1]) # "banana"**

**2)Mutable**: Lists are mutable, which means you can modify them by adding, removing, or changing elements after they have been created. For example:

**Program:-**

**fruits = ["apple", "banana", "orange"]**

**fruits.append("grape") # add an element to the end of the list**

**fruits[1] = "pear" # change the second element**

**del fruits[0] # remove the first element**

**3)Flexible**: Lists can contain elements of any data type, including other lists (nested lists). For example:

**Program:-**

**mixed\_list = [1, "apple", True, [2, 3, 4]]**

**4)Iterable:** Lists are iterable, which means you can loop over the elements in a list using a for loop. For example:

**Program:-**

**fruits = ["apple", "banana", "orange"]**

**for fruit in fruits:**

**print(fruit)**

**5)Length:** You can use the len() function to get the number of elements in a list. For example:

**Program:-**

**fruits = ["apple", "banana", "orange"]**

**print(len(fruits)) # 3**

Lists are a versatile and commonly used data structure in Python, and they provide a convenient way to work with collections of data.

**11. What is Feature of Tuple?**

In Python, a tuple is an ordered and immutable collection of elements, which means that once a tuple is created, its contents cannot be changed. Here are some of the main features of tuples:

Ordered: Tuples are ordered, which means that the elements in a tuple have a specific order, and that order will not change.

**Immutable:** Tuples are immutable, which means that once a tuple is created, its contents cannot be modified. This makes tuples useful for storing values that should not be changed, such as configuration settings or constants.

Can contain any data type: Tuples can contain any data type, including numbers, strings, lists, dictionaries, and other tuples.

**Indexed:** Tuples are indexed, which means that you can access individual elements in a tuple by their index.

**Faster than lists:** Tuples are generally faster than lists for certain operations, such as accessing elements by index or iterating over the elements.

Can be used as keys in dictionaries: Because tuples are immutable, they can be used as keys in dictionaries, whereas lists cannot.

Here's an example of creating a tuple:

**Program:-**

**my\_tuple = (1, "two", 3.0, [4, 5, 6])**

In this example, my\_tuple is a tuple that contains four elements: an integer, a string, a floating-point number, and a list.

Overall, tuples are a useful data structure in Python for storing ordered, immutable collections of data.

**12. What is Feature of Set?**

In Python, a set is an unordered collection of unique elements. It is defined by enclosing a comma-separated list of elements within curly braces { }, or by using the set() function to convert a list or tuple to a set.

The key features of a set in Python are:

**1.Uniqueness:** A set can only contain unique elements. If you try to add an element that already exists in the set, it will not be added.

**2.Unorderedness:** The elements in a set are not stored in any specific order. When you iterate over a set, the order in which you get the elements may vary.

**3.Mutability:** A set is mutable, which means you can add or remove elements from it.

**4.Hashability:** The elements in a set must be hashable. This means that the elements must have a hash value that does not change during their lifetime. Immutable types like numbers, strings, and tuples are hashable, but mutable types like lists and dictionaries are not.

**5.Set operations:** Sets support various set operations such as union, intersection, difference, symmetric difference, and subset/superset tests. These operations can be performed using methods or operators like |, &, -, ^, and <=/>=.

Here's an example that demonstrates some of these features:

**Program:-**

**# define a set**

**fruits = {'apple', 'banana', 'orange', 'banana'}**

**# print the set (note that duplicates are removed)**

**print(fruits) # {'orange', 'apple', 'banana'}**

**# add an element to the set**

**fruits.add('grape')**

**print(fruits) # {'orange', 'grape', 'banana', 'apple'}**

**# remove an element from the set**

**fruits.remove('banana')**

**print(fruits) # {'orange', 'grape', 'apple'}**

**# set operations**

**a = {1, 2, 3, 4}**

**b = {3, 4, 5, 6}**

**print(a | b) # {1, 2, 3, 4, 5, 6}**

**print(a & b) # {3, 4}**

**print(a - b) # {1, 2}**

**print(b - a) # {5, 6}**

**print(a ^ b) # {1, 2, 5, 6}**

Sets are commonly used in Python for various purposes such as removing duplicates, testing membership, and performing set operations.

**13. What is feature of Frozen Set?**

In Python, a frozenset is an immutable collection of unique elements. Once a frozenset is created, its elements cannot be modified, added or removed. The main feature of a frozenset is its immutability, which makes it suitable for use as a dictionary key or an element of another set, since these data types also require immutable elements.

Here are some features of frozenset in Python:

**1.Immutable:** Once a frozenset is created, its elements cannot be modified, added, or removed. This makes it a hashable and immutable object that can be used as a dictionary key or an element of another set.

**2.Unique elements:** A frozenset can only contain unique elements. If you try to create a frozenset with duplicate elements, the duplicates will be eliminated automatically.

**3.Set operations:** frozenset supports set operations such as union, intersection, difference, and symmetric difference. These operations return a new frozenset object and do not modify the original frozenset.

**4.Type conversion:** A frozenset can be created from an iterable such as a list, tuple, or another set using the frozenset() constructor. Similarly, a frozenset can be converted back to a regular set using the set() constructor.

Here's an example that demonstrates the features of frozenset:

**Program:-**

**# create a frozenset**

**fs = frozenset([1, 2, 3, 4, 5])**

**# print the frozenset**

**print(fs) # frozenset({1, 2, 3, 4, 5})**

**# try to add an element (raises an AttributeError)**

**try:**

**fs.add(6)**

**except AttributeError as e:**

**print(e) # 'frozenset' object has no attribute 'add'**

**# set operations**

**s1 = frozenset([1, 2, 3])**

**s2 = frozenset([3, 4, 5])**

**print(s1.union(s2)) # frozenset({1, 2, 3, 4, 5})**

**print(s1.intersection(s2)) # frozenset({3})**

**print(s1.difference(s2)) # frozenset({1, 2})**

**print(s1.symmetric\_difference(s2)) # frozenset({1, 2, 4, 5})**

In this example, we create a frozenset from a list, and then we try to add an element to the frozenset (which raises an AttributeError since frozenset is immutable). We also demonstrate some set operations that return a new frozenset object.

**14. What is Feature of Dictionary?**

In Python, a dictionary is an unordered collection of key-value pairs, where each key is unique. Dictionaries are very useful data structures, and they have the following features:

**1.Mutable:** Dictionaries are mutable, which means you can add, modify, or remove key-value pairs after the dictionary is created.

**2.Unordered:** Dictionaries are unordered, which means the order of key-value pairs is not fixed or guaranteed. This is different from lists, which are ordered.

**3.Indexed by keys:** Dictionaries are indexed by keys, not by positions. You can use any hashable object as a key, such as strings, numbers, and tuples, but you cannot use mutable objects such as lists as keys.

**4.Flexible data types:** Dictionaries can store values of different data types, such as strings, numbers, lists, tuples, and even other dictionaries.

**5.Fast lookup:** Dictionaries provide fast lookup time for retrieving the value associated with a given key. This is because dictionaries use hash tables to store and retrieve key-value pairs efficiently.

Here's an example of creating a dictionary in Python:

**Program:-**

**# create a dictionary of person's information**

**person = {**

**"name": "John",**

**"age": 30,**

**"city": "New York"**

**}**

**# access a value by key**

**print(person["name"]) # John**

**# add a new key-value pair**

**person["email"] = "john@example.com"**

**# modify a value**

**person["age"] = 31**

**# remove a key-value pair**

**del person["city"]**

Dictionaries are commonly used in many Python programs, such as web applications, data analysis, and machine learning, where they are used to store and manipulate data efficiently.

**15. What is Type Casting?**

Type casting, also known as type conversion, is the process of converting a variable of one data type to another data type. In Python, you can use built-in functions to perform type casting.

Here are some of the built-in functions for type casting in Python:

1.int(): converts a variable to an integer data type

2.float(): converts a variable to a float data type

3.str(): converts a variable to a string data type

4.bool(): converts a variable to a boolean data type

Here's an example of how to use type casting in Python:

**Program:-**

**# convert a string to an integer**

**age\_str = "30"**

**age\_int = int(age\_str)**

**# convert an integer to a string**

**count\_int = 10**

**count\_str = str(count\_int)**

**# convert a float to an integer**

**price\_float = 9.99**

**price\_int = int(price\_float)**

**# convert an integer to a boolean**

**num = 5**

**is\_greater\_than\_3 = bool(num > 3)**

Type casting is important when you need to perform operations or comparisons between variables of different data types. It allows you to convert a variable to the appropriate data type before performing the operation, and it can help prevent errors and ensure correct results.

**16.what is escape sequence?**

An escape sequence in Python is a sequence of characters that represents a special meaning when used inside a string. Escape sequences start with a backslash (\) followed by one or more characters. The backslash tells Python to interpret the following characters in a special way.

Here are some commonly used escape sequences in Python:

\n: newline character

\t: tab character

\\: backslash character

\': single quote character

\": double quote character

Here's an example of how to use escape sequences in Python:

**Program:-**

**# newline and tab characters**

**print("Hello\nWorld!")**

**print("Name:\tJohn")**

**# backslash and quote characters**

**print("This is a backslash: \\")**

**print("He said, \"I'm going to the store.\"")**

In the above example, the \n escape sequence is used to create a new line, and the \t escape sequence is used to insert a tab. The \\ escape sequence is used to insert a backslash, and the \' and \" escape sequences are used to insert single and double quotes, respectively.

Escape sequences are useful for inserting special characters inside strings, and they allow you to create more complex and flexible output in your Python programs.

**17. What is String Slicing?**

In Python, string slicing is the process of extracting a subset of characters from a string. You can use slicing to extract a range of characters from a string based on their positions or indices.

The syntax for string slicing is as follows:

**Program:-**

**string[start:stop:step]**

start: The starting index of the slice (inclusive). If omitted, the slice starts from the beginning of the string.

stop: The ending index of the slice (exclusive). If omitted, the slice extends to the end of the string.

step: The step size between characters in the slice. If omitted, the default value is 1.

Here's an example of how to use string slicing in Python:

**Program:-**

**# define a string**

**s = "Hello, World!"**

**# extract a subset of characters using slicing**

**substring1 = s[0:5] # "Hello"**

**substring2 = s[7:] # "World!"**

**substring3 = s[:5] # "Hello"**

**substring4 = s[::2] # "Hlo ol!"**

**# print the substrings**

**print(substring1)**

**print(substring2)**

**print(substring3)**

**print(substring4)**

In the example above, the first slice (substring1 = s[0:5]) extracts the characters from position 0 to position 4 (inclusive), which is the substring "Hello". The second slice (substring2 = s[7:]) extracts all characters from position 7 to the end of the string, which is the substring "World!". The third slice (substring3 = s[:5]) extracts the characters from the beginning of the string up to position 4 (inclusive), which is the substring "Hello". The fourth slice (substring4 = s[::2]) extracts every other character from the string, starting from the beginning and skipping one character at a time, which is the substring "Hlo ol!".

String slicing is a useful technique for manipulating strings in Python, and it can be used in a variety of string operations, such as searching, replacing, and formatting.

**18.What is Conditional Statement?**

A conditional statement, also known as a conditional expression, is a programming construct that allows you to control the flow of a program based on a condition. In Python, there are two types of conditional statements: the if statement and the if-else statement.

The if statement is used to execute a block of code only if a certain condition is true. Here's an example:

**Program:-**

**x = 10**

**if x > 5:**

**print("x is greater than 5")**

In this example, the if statement checks if x is greater than 5. If the condition is true, the print() statement is executed.

The if-else statement is used to execute one block of code if a condition is true, and a different block of code if the condition is false. Here's an example:

**Program:-**

**x = 3**

**if x > 5:**

**print("x is greater than 5")**

**else:**

**print("x is less than or equal to 5")**

In this example, the if statement checks if x is greater than 5. If the condition is true, the first print() statement is executed. If the condition is false, the else block is executed, and the second print() statement is executed.

Conditional statements are very useful in programming, as they allow you to make decisions based on certain conditions. They can be used in a wide variety of applications, such as user input validation, error handling, and program flow control.

**18. What is Datatype in Python?**

In Python, a data type is a classification of the type of data that a variable or expression can hold. Python has several built-in data types, such as integers, floating-point numbers, strings, and Booleans.

Here are some examples of data types in Python:

1.Integers: whole numbers without a decimal point, such as 1, 2, 3, -4, -5, etc.

2.Floats: numbers with decimal points, such as 3.14, -0.23, 2.0, etc.

3.Strings: sequences of characters, such as "hello", "world", "123", etc.

4.Booleans: a data type that represents either True or False.

Python also has other built-in data types such as lists, tuples, dictionaries, and sets. Additionally, Python allows users to define their own custom data types using classes.

**19. What are the type of Datatype? What is Types of Looping Statement ?**

In Python, there are several built-in data types, including:

**Numeric types:**

* Integer (int): whole numbers such as 1, 2, 3, etc.
* Float (float): decimal or floating-point numbers such as 3.14, 2.0, etc.
* Complex (complex): numbers in the form a + bj, where a and b are floats, and j represents the imaginary unit.

Text types:

String (str): sequence of characters such as "hello", "world", etc.

Sequence types:

List (list): ordered collection of items, such as [1, 2, 3], ["apple", "banana", "orange"], etc.

Tuple (tuple): ordered, immutable collection of items, such as (1, 2, 3), ("apple", "banana", "orange"), etc.

Range (range): an immutable sequence of numbers.

Mapping types:

Dictionary (dict): unordered collection of key-value pairs, such as {"name": "John", "age": 30}, etc.

Set types:

Set (set): unordered collection of unique items, such as {1, 2, 3}, {"apple", "banana", "orange"}, etc.

Frozen set (frozenset): an immutable version of set.

Boolean types:

Boolean (bool): represents True or False.

Binary types:

Bytes (bytes): a immutable sequence of bytes, such as b"hello", b"\x01\x02\x03", etc.

Bytearray (bytearray): a mutable version of bytes.

**20. What is Looping Statement?**

In Python, a looping statement allows you to execute a block of code repeatedly. There are two main types of looping statements in Python: "for" loop and "while" loop.

"for" loop:

The "for" loop is used to iterate over a sequence (such as a list, tuple, or string) or other iterable object, and execute a block of code for each item in the sequence. The general syntax of a "for" loop is:

for variable in sequence:

# execute code here

Here, "variable" represents the current item in the sequence, and "sequence" is the sequence that we want to iterate over.

Example:

**Program:-**

**fruits = ["apple", "banana", "cherry"]**

**for fruit in fruits:**

**print(fruit)**

**Output:**

**apple**

**banana**

**cherry**

2."while" loop:

The "while" loop is used to execute a block of code repeatedly as long as a certain condition is true. The general syntax of a "while" loop is:

while condition:

# execute code here

Here, "condition" is a boolean expression that determines whether or not to continue looping.

**Example:**

**count = 0**

**while count < 5:**

**print(count)**

**count += 1**

**Output:-**

**0**

**1**

**2**

**3**

**4**

In addition to the basic "for" and "while" loops, Python also provides some additional looping statements, such as "break" and "continue", which can be used to control the flow of the loop.

**21. When to use which type of conditional statement?**

In Python, there are two main types of conditional statements: "if" statement and "switch-case" statement is not available in Python.

**1."if" statement:**

**The "if" statement is used to execute a block of code only if a certain condition is true. The general syntax of an "if" statement is:**

**if condition:**

**# execute code here**

Here, "condition" is a boolean expression that determines whether or not to execute the code block.

Use "if" statement when you want to execute a block of code only if a certain condition is true. For example, if you want to check if a number is positive or negative, you can use an "if" statement:

**Program:-**

**num = -5**

**if num >= 0:**

**print("The number is positive")**

**else:**

**print("The number is negative")**

Output:

The number is negative

2."switch-case" statement:

Unlike some other programming languages, Python does not have a built-in "switch-case" statement. However, you can achieve similar functionality using "if-elif-else" statements. The general syntax of an "if-elif-else" statement is:

**Program:**

**if condition1:**

**# execute code here**

**elif condition2:**

**# execute code here**

**elif condition3:**

**# execute code here**

**...**

**else:**

**# execute code here**

Here, "condition1", "condition2", etc. are boolean expressions that are evaluated in order. When a condition is true, the corresponding code block is executed, and the rest of the conditions are not evaluated.

Use "if-elif-else" statement when you have multiple conditions to check and want to execute a different block of code for each condition. For example, if you want to assign a letter grade based on a student's score, you can use an "if-elif-else" statement:

**Program:**

**score = 85**

**if score >= 90:**

**grade = "A"**

**elif score >= 80:**

**grade = "B"**

**elif score >= 70:**

**grade = "C"**

**elif score >= 60:**

**grade = "D"**

**else:**

**grade = "F"**

**print("The grade is:", grade)**

**Output:**

**The grade is: B**

**22. When to use which type of looping statement?**

In Python, there are two types of looping statements: "for" loop and "while" loop.

1."for" loop:

A "for" loop is used to iterate over a sequence of items. The general syntax of a "for" loop is:

for item in sequence:

# execute code here

Here, "item" is a variable that takes on the value of each item in the sequence, and "sequence" is the sequence of items to iterate over.

Use a "for" loop when you want to execute a block of code for each item in a sequence. For example, if you want to print each character in a string, you can use a "for" loop:

**Program:**

**str = "hello"**

**for char in str:**

**print(char)**

**Output:**

**h**

**e**

**l**

**l**

**o**

2."while" loop:

The "while" loop is used to execute a block of code repeatedly as long as a certain condition is true. The general syntax of a "while" loop is:

while condition:

# execute code here

Here, "condition" is a boolean expression that determines whether or not to continue looping.

Use a "while" loop when you want to execute a block of code repeatedly until a certain condition is met. For example, if you want to keep adding numbers to a sum until the sum exceeds a certain value, you can use a "while" loop:

**Program:**

**sum = 0**

**num = 1**

**while sum < 100:**

**sum += num**

**num += 1**

**print("The sum is:", sum)**

**Output:**

**The sum is: 105**

In general, use a "for" loop when you know the number of times you want to execute a block of code (i.e., you have a sequence to iterate over), and use a "while" loop when you don't know the number of times you want to execute a block of code (i.e., you are waiting for a certain condition to be met).

23.What is Function in python?

A function in Python is a block of reusable code that performs a specific task. It can be used to group related code together, improve code readability and reusability, and reduce the amount of duplicated code in a program. In Python, you can create a function using the "def" keyword, followed by the function name, a set of parentheses, and a colon. Here's the general syntax of a function definition:

Program:

def function\_name(parameters):

"""docstring - optional"""

# code block

return expression

Here, "function\_name" is the name of the function, and "parameters" are optional variables that are passed to the function as input. The docstring is an optional string that describes what the function does. The code block contains the instructions that are executed when the function is called. The "return" statement is optional and is used to return a value from the function.

To call a function in Python, you simply use its name followed by a set of parentheses containing any necessary arguments. Here's an example of calling a function:

Program:

def add\_numbers(x, y):

"""Adds two numbers and returns the result"""

return x + y

result = add\_numbers(3, 5)

print(result)

Output:

8

In this example, we defined a function called "add\_numbers" that takes two parameters and returns their sum. We then called the function with arguments 3 and 5, and stored the result in a variable called "result". Finally, we printed the value of "result", which is 8.

24. What are type of Arguments we can pass in python function?

In Python, there are four types of arguments that can be passed to a function:

1.Positional arguments: These are the most common type of argument and are passed based on their position in the function call. The first argument is assigned to the first parameter, the second argument is assigned to the second parameter, and so on. Example:

Program:

def greet(name, greeting):

print(greeting + ", " + name + "!")

greet("Alice", "Hello")

Output:

Hello, Alice!

2.Keyword arguments: These are passed with a name-value pair and are assigned to the corresponding parameter based on the parameter name. Keyword arguments can be used in any order, and can be mixed with positional arguments. Example:

Program:

def greet(name, greeting):

print(greeting + ", " + name + "!")

greet(greeting="Hello", name="Alice")

Output:

Hello, Alice!

3.Default arguments: These are used to assign a default value to a parameter, in case no value is provided in the function call. Default arguments are specified in the function definition. Example:

Program:

def greet(name, greeting="Hello"):

print(greeting + ", " + name + "!")

greet("Alice")

Output:

Hello, Alice!

4.Variable-length arguments: These are used when you don't know how many arguments will be passed to the function. There are two types of variable-length arguments in Python:

\*args: This is used to pass a variable-length list of positional arguments to a function. The arguments are passed as a tuple. Example:

Program:

def greet\_all(\*names):

for name in names:

print("Hello, " + name + "!")

greet\_all("Alice", "Bob", "Charlie")

Output:

Hello, Alice!

Hello, Bob!

Hello, Charlie!

\*\*kwargs: This is used to pass a variable-length dictionary of keyword arguments to a function. The arguments are passed as a dictionary. Example:

Program:

def greet\_all(\*\*names\_greetings):

for name, greeting in names\_greetings.items():

print(greeting + ", " + name + "!")

greet\_all(Alice="Hello", Bob="Hi", Charlie="Hey")

Output:

Hello, Alice!

Hi, Bob!

Hey, Charlie!

25. What is Modules?

In Python, a module is a file that contains a set of related functions, variables, and classes. Modules are used to organize code into logical units and to avoid naming collisions between different code blocks. A module can be imported into another Python script or module, allowing the code inside it to be reused and shared.

In Python, a module is simply a Python file with a .py extension. The file name becomes the module name, and the code inside the file defines the contents of the module. Here's an example of a simple module:

Program:

# my\_module.py

def greet(name):

print("Hello, " + name + "!")

def add\_numbers(x, y):

return x + y

To use the functions defined in a module, you need to import it into your code. There are several ways to import a module in Python:

1.import statement: This is the most common way to import a module. It imports the entire module, and you can access the functions and variables in the module using dot notation. Example:

Program:

import my\_module

my\_module.greet("Alice")

Output:

Hello, Alice!

2.from-import statement: This allows you to import specific functions or variables from a module, rather than importing the entire module. Example:

Program:

from my\_module import greet

greet("Alice")

Output:

Hello, Alice!

3.import-as statement: This allows you to import a module or function with a different name, to avoid naming collisions. Example:

Program:

import my\_module as mm

mm.greet("Alice")

Output:

Hello, Alice!

26. What is Package?

In Python, a package is a way to organize related modules into a single namespace or directory hierarchy. A package can contain multiple modules and sub-packages, and can be used to create a hierarchical structure for your code.

Packages are similar to directories in a file system, and can contain other packages, as well as Python modules and files. Packages are defined by creating a directory with an init.py file inside it. The init.py file can be empty or can contain Python code, and is executed when the package is imported.

Here's an example of a simple package with a module inside it:

Program1:

# my\_package/\_\_init\_\_.py

print("Importing my\_package...")

\_\_all\_\_ = ['my\_module']

Program2:

# my\_package/my\_module.py

def greet(name):

print("Hello, " + name + "!")

To use the module in the package, you can import it using the package name and module name separated by a dot. For example:

Program:

import my\_package.my\_module

my\_package.my\_module.greet("Alice")

Output:

Importing my\_package...

Hello, Alice!

You can also use the from-import statement to import the module directly, like this:

Program:

from my\_package import my\_module

my\_module.greet("Alice")

Output:

Importing my\_package...

Hello, Alice!

27. What is Class?

In Python, a class is a blueprint for creating objects that have certain attributes (properties) and methods (functions). It provides a way to create new user-defined data types and define their behavior.

A class defines a set of attributes that are common to all objects of that class. These attributes are defined using variables called instance variables, which are unique to each instance of the class. The class also defines methods that can be called on the objects of that class. These methods can manipulate the instance variables and perform other operations on the object.

Here's an example of a simple class definition in Python:

program:

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def greet(self):

print("Hello, my name is " + self.name + " and I am " + str(self.age) + " years old.")

This class defines a Person object with two instance variables (name and age) and one method (greet). The \_\_init\_\_ method is a special method that is called when a new object of the class is created, and is used to initialize the instance variables.

To create an object of the Person class, you can call the class constructor with the required arguments, like this:

Program:

person = Person("Alice", 30)

This creates a new Person object with name "Alice" and age 30. You can then call the greet method on the object, like this:

Program:

person.greet()

Output:

Hello, my name is Alice and I am 30 years old.

28. What is Object?

In Python, an object is an instance of a class that has its own unique set of attributes (properties) and methods (functions). Objects are created from classes using the class constructor, and each object has its own state (values of its instance variables) and behavior (methods).

Objects are often used to represent real-world entities or concepts in a program. For example, a Person class can be used to represent a person, with attributes like name, age, and address, and methods like walk and talk. Each Person object created from the class would have its own set of attribute values and can perform its own actions based on its methods.

Here's an example of creating an object of a class in Python:

program:

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def greet(self):

print("Hello, my name is " + self.name + " and I am " + str(self.age) + " years old.")

person1 = Person("Alice", 30)

person2 = Person("Bob", 25)

person1.greet() # Output: Hello, my name is Alice and I am 30 years old.

person2.greet() # Output: Hello, my name is Bob and I am 25 years old.

In this example, we create two objects (person1 and person2) of the Person class, each with its own set of attribute values (name and age) and behavior (greet method). When we call the greet method on each object, it prints a message with the object's attribute values.

29. Why to create Object?

Creating objects is a fundamental concept in object-oriented programming (OOP) and it provides a number of benefits:

1.Encapsulation: Objects allow you to encapsulate related data and methods into a single entity. This makes it easier to manage and organize your code, and helps to prevent naming conflicts.

2.Reusability: Once you create an object, you can reuse it in multiple parts of your code without having to rewrite the same code again and again.

3.Modularity: Objects can be easily added, removed, or modified without affecting other parts of your code. This makes your code more modular and easier to maintain.

4.Abstraction: Objects allow you to abstract away complex details of your code, making it easier to understand and work with.

5.Polymorphism: Objects can have multiple forms or behaviors based on the context in which they are used. This allows you to write more flexible and reusable code.

Overall, creating objects allows you to write cleaner, more organized, and more maintainable code, and it is a fundamental concept in OOP that is used extensively in Python and other programming languages.

30. What is Inner class?

In Python, an inner class is a class that is defined inside another class. Inner classes are sometimes also called nested classes.

The inner class can access the attributes and methods of the outer class, which can be useful in some situations. For example, an inner class can be used to define a helper class that is closely related to the outer class.

Here's an example of an inner class in Python:

program:

class Outer:

def \_\_init\_\_(self, x):

self.x = x

self.inner = self.Inner()

def print\_inner\_x(self):

print(self.inner.x)

class Inner:

def \_\_init\_\_(self):

self.x = 10

In this example, we define an outer class Outer with an inner class Inner. The inner class has its own \_\_init\_\_ method, which initializes its own instance variable x.

To create an instance of the inner class, we create an instance of the outer class and then access the inner class using the dot notation, like this:

program:

outer = Outer(5)

inner = outer.inner

We can then access the instance variable x of the inner class using the dot notation, like this:

Program:

print(inner.x) # Output: 10

Alternatively, we can access the x variable of the inner class through a method of the outer class that uses the inner class, like this:

program:

outer.print\_inner\_x() # Output: 10

Note that because the inner class is defined inside the outer class, its name is actually scoped to the outer class. To access the inner class from outside the outer class, you need to use the full name of the inner class, like this:

program:

inner = Outer.Inner()

31. How many ways are there to create Objects of inner class?

In Python, there are two ways to create objects of an inner class:

1.Using the outer class instance: You can create an instance of the inner class by calling it as an attribute of an instance of the outer class. For example, consider the following code:

program:

class Outer:

def \_\_init\_\_(self):

self.inner = self.Inner()

class Inner:

def \_\_init\_\_(self):

self.message = "Hello, world!"

outer = Outer()

inner = outer.inner

print(inner.message) # Output: "Hello, world!"

2.In this example, we create an instance of the Outer class and then access its inner attribute to create an instance of the Inner class. We can then access the message attribute of the inner instance using the inner variable.

Using the outer class name: You can also create an instance of the inner class by calling it as an attribute of the outer class itself. For example:

Program:

class Outer:

def \_\_init\_\_(self):

self.inner = self.Inner()

class Inner:

def \_\_init\_\_(self):

self.message = "Hello, world!"

inner = Outer.Inner()

print(inner.message) # Output: "Hello, world!"

In this example, we create an instance of the Inner class by calling it as an attribute of the Outer class itself. We can then access the message attribute of the inner instance using the inner variable.

Note that in both cases, the inner class is accessed using the dot notation, either as an attribute of an instance of the outer class or as an attribute of the outer class itself.

32. What is Constructor?

In Python, a constructor is a special method that is called when an object is created. It is used to initialize the object's attributes, allocate memory for the object, and perform any other setup required by the object.

The constructor method in Python is always named \_\_init\_\_. It takes at least one argument, which is usually named self. The self argument refers to the instance of the object being created.

Here's an example of a simple class with a constructor:

Program:

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

In this example, we define a Person class with a constructor that takes two arguments: name and age. The constructor initializes the name and age attributes of the object using the values of the arguments.

To create an instance of the Person class, we call the class as if it were a function, passing in the required arguments:

program:

person = Person("Alice", 25)

In this example, we create an instance of the Person class with the name "Alice" and the age 25. The \_\_init\_\_ method is automatically called to initialize the name and age attributes of the person object.

Note that the self argument is not explicitly passed in when we create an object of the class. Python takes care of passing the instance of the object to the constructor automatically, so we don't need to worry about it.

33. What is Garbage Collector?

In Python, garbage collection is the process of automatically freeing memory that is no longer being used by the program. The garbage collector is a component of the Python interpreter that runs in the background and identifies memory that is no longer being used by the program, and frees it up for reuse.

When an object is no longer referenced by any part of the program, it becomes eligible for garbage collection. The garbage collector then takes care of deallocating the memory that was used by the object.

Python's garbage collector uses a technique called reference counting to determine when an object is no longer being used. Each object in Python has a reference count, which is the number of references to the object. When the reference count of an object drops to zero, it means that the object is no longer being used, and it can be freed up.

In addition to reference counting, Python also uses a technique called cycle detection to handle circular references. Circular references occur when two or more objects reference each other, creating a cycle. Python's garbage collector can detect these cycles and free up the memory used by the objects in the cycle.

In general, Python's garbage collector is very effective at managing memory automatically, so you don't have to worry about deallocating memory yourself. However, it's important to be aware of how memory management works in Python, especially if you're working with large data sets or long-running programs that might consume a lot of memory over time.

34. What is Destructor?

In Python, a destructor is a special method that is called when an object is destroyed. The destructor is responsible for freeing up any resources that were allocated by the object during its lifetime, such as closing files or releasing memory.

In Python, the destructor method is called \_\_del\_\_. It takes one argument, which is usually named self, and is automatically called by the Python interpreter when the object is no longer being used and is ready to be destroyed.

Here's an example of a simple class with a destructor:

Program:

class MyClass:

def \_\_init\_\_(self):

print("Creating object")

def \_\_del\_\_(self):

print("Destroying object")

In this example, we define a MyClass class with a constructor that prints a message when an object is created, and a destructor that prints a message when the object is destroyed.

To create an instance of the MyClass class, we simply call the class as if it were a function:

obj = MyClass()

In this example, we create an instance of the MyClass class and store it in the variable obj. The constructor is automatically called, which prints the message "Creating object".

When the program exits or when the object is no longer being used, the destructor is automatically called, which prints the message "Destroying object".

Note that you generally don't need to define a destructor in Python, because the garbage collector takes care of freeing up memory automatically when objects are no longer being used. However, if you need to release other resources besides memory, such as closing files or network connections, you may want to define a destructor to handle those tasks.

35. What is Inheritance?

Inheritance is a mechanism in object-oriented programming (OOP) that allows a new class (called the derived class) to be based on an existing class (called the base or parent class). The derived class inherits all the methods and properties of the base class, and can also add new methods and properties of its own.

Inheritance allows you to create new classes that are similar to existing classes, but with some differences. For example, you might have a base class called Vehicle that defines properties like color, weight, and price, as well as methods like start\_engine() and stop\_engine(). You could then create a derived class called Car that inherits all these properties and methods from Vehicle, but also adds its own properties like num\_doors and num\_wheels, and methods like change\_gear() and apply\_brakes().

In Python, you can define a derived class by specifying the base class in parentheses after the class name. For example:

program:

class Vehicle:

def \_\_init\_\_(self, color, weight, price):

self.color = color

self.weight = weight

self.price = price

def start\_engine(self):

print("Engine started")

def stop\_engine(self):

print("Engine stopped")

class Car(Vehicle):

def \_\_init\_\_(self, color, weight, price, num\_doors, num\_wheels):

super().\_\_init\_\_(color, weight, price)

self.num\_doors = num\_doors

self.num\_wheels = num\_wheels

def change\_gear(self, gear):

print("Changing gear to", gear)

def apply\_brakes(self):

print("Applying brakes")

In this example, we define a base class called Vehicle with an \_\_init\_\_() method and two other methods. We then define a derived class called Car that inherits from Vehicle and adds its own \_\_init\_\_() method and two other methods.

To create an instance of the Car class, we can simply call the class as if it were a function, passing in the required arguments:

my\_car = Car("red", 2000, 10000, 4, 4)

In this example, we create an instance of the Car class with the specified color, weight, price, number of doors, and number of wheels. The \_\_init\_\_() method of the Vehicle class is automatically called to initialize the color, weight, and price properties of the Car object, and the \_\_init\_\_() method of the Car class initializes the num\_doors and num\_wheels properties.

37. What are different types of Inheritance and explain each one of them?

38. What is polymorphism?

39. Does Python Supports Polymorphism? If Not why?

40. How to achieve Polymorphism in python?

41. What is Duck typing in python?

42. What is Abstraction?

43. What is Encapsulation?

44. How to call private method outside class?

45. How to call protected method outside the class?

46. What is file handling in python?

50. What is exception Handling?

51. Define All list built-in function .

52. Define all tuple built-in function

53. Define all set built-in function

54. Define all Dict built-in function

55. What is Transfer Statement?

56. What is difference between break and continue?